Paul J. Conrad
CS 610, Winter 2004
Dr. Keith Schubert

The Concise Cache Tutorial

Table of Contents

Part I. Cache Introduction

What is a cache? A cache is defined as a small, but fast memory holding recently accessed data, which is designed to speed up subsequent access to the same data. Frequently accessed data such as machine instructions, or data values within a computer program, can be accessed at a faster rate in the cache, rather than from the main memory. This is for several different reasons that will be covered in this tutorial. Figure 1 shows a typical multilevel memory hierarchy in which the faster, expensive memories are closer to the top of the triangle.

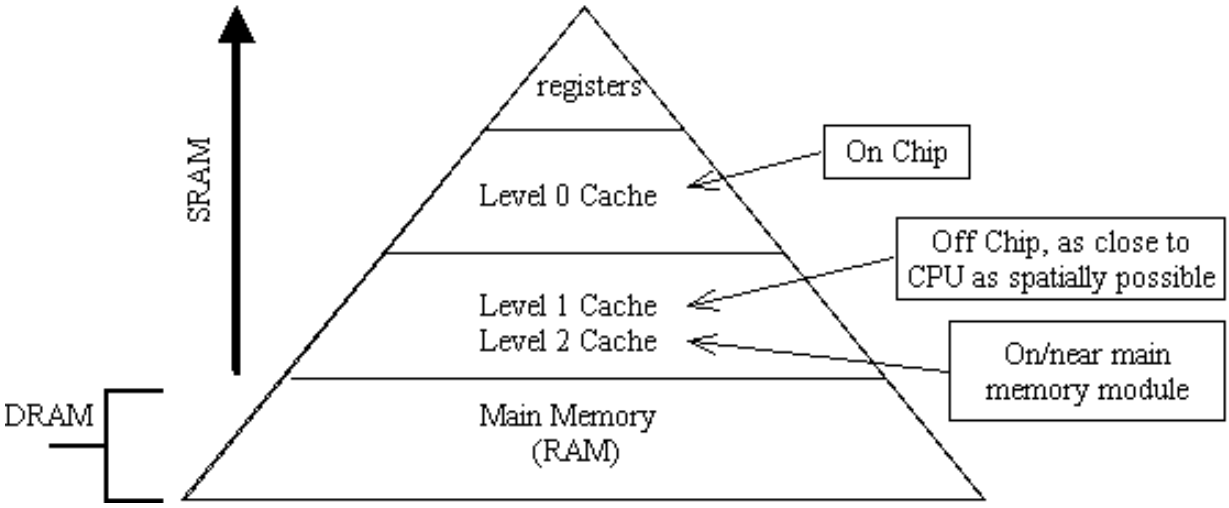


Figure 1. A Typical Memory Hierarchy

The registers are, of course, found inside the CPU because they work directly with the CPU. The Level 0 Cache is generally located on the CPU chip between the CPU and the Level 1 Cache. The Level 1 Cache sits between the Level 1 Cache and the Level 2 Cache, which is found on, or near the main memory modules. The registers are generally running at speeds of 0.25-1 ns, and have a total size of ≈72-500 bytes. As we traverse down the hierarchy towards the main memory, we decrease in speed and increase in the memory capacity. In Figure 2, we have a broader memory hierarchy that is not as low level as in Figure 1. We have our first box on the far left side representing the actual CPU, as we move to the right, we run into the Level 2 Cache. These continues on along the memory bus to the main memory, and the main memory, and then go on to the I/O Devices such as the hard disk to utilize Virtual Memory.

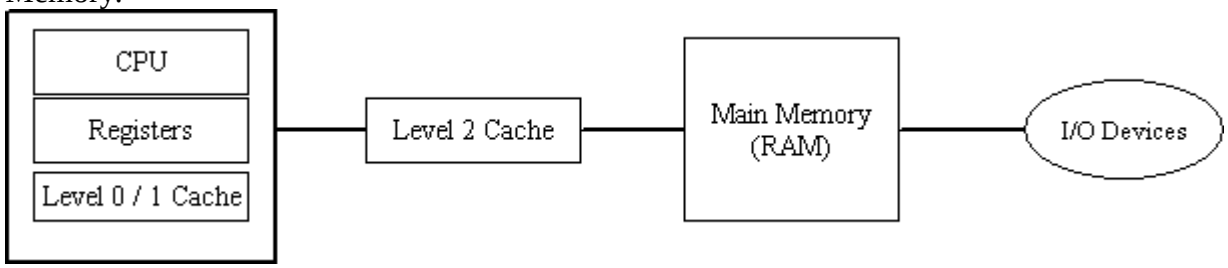


Figure 2. Higher Level Memory Hierarchy

Importance of Cache in Computer Science

Why is cache so important in Computer Science? Cache allows for the performance of a computer program to increase depending on how well the software engineer and compiler writer knows about the underlying hardware architecture that the generated executable program is going to be running on. Since software engineering and computer programming is not exactly a magical art that is confined to a select few running around looking like Merlin the wizard, just about anyone can be a computer programmer. These programmers can either be "just a programmer," or they can be an "expert programmer" who takes a closer look at the underlying architecture in which the software being developed is targeted for. These expert programmers write more than just computer programs, they write very good pieces of code that take full use of the hardware's features.

In order for a software engineer or programmer to be able to fully utilize the hardware, the features of the hardware should be well understood. These features, to just name some, are: cache, pipelining, loop-unrolling optimizations, and various other code optimization techniques to take full advantage of the hardware. This tutorial is going to be focusing on understanding the cache, how to utilize the cache, measuring of cache performance, and how to improve cache performance.

Cache Tutorial Test Benches

Through out this tutorial, there are going to be some example runs of test bench code snippets that are before and after that will demonstrate the impacts that the code improvements have when utilizing the cache. The programs have been compiled with Microsoft® Visual C++™ and run from Windows XP Pro™.

The test bench codes in this tutorial are used to demonstrate the effective performance increase using *loop interchanging*, *matrix multiplication via cache blocking*, and a *simple loop unrolling* approach. As mentioned, the programs were compiled using the free Visual C++™ command line compiler that is available from Microsoft®. The programs were compiled using the following compiler options:

```
/O2         Maximize speed
/Ox         Maximize optimizations
/Ob2        Inline expansion (set n=2)
/GS         Enable security checks
/GL         Enable link-time code generation
/arch:SSE2  Enable SSE2 instructions since Pentium 4 was used
```

In Appendix A, the source code to the programs are available, along with batch script files that were used to generate the executables and a script file that ran each of these tests.

The programs all have the ability to generate a CSV (comma separated value) text file of the instance size, before clock cycle reading, and after clock cycle reading. The tests ran each instance seven (7) times and when analysis was done, the minimum read, maximum read were removed and the average was taken of the remaining five (5) reads.

Part II. Cache Basics

Common Terminologies Used

Since now we have an idea of the purpose of having cache in computer systems, there are some important cache terms or buzzwords that may appear from time to time in this tutorial. Again, since this tutorial is directed towards graduate students looking to supplement Hennessey and Patterson's [1] book, it can be also directed to advanced undergraduate students taking an advanced computer architecture course. Here are most of the basic terms from Hennessey and Patterson [1], and some in easier definition:

- **Address trace** – the tracing of instruction and data references to measure cache miss rates with cache simulators.

- **Average memory access time** – measure of the memory hierarchy performance, where it is computed as:

  hit time + miss rate * miss penalty.

  Hit time is the amount of time spent when a cache hit occurs. The miss rate is the frequency in which a cache miss occurs, and the miss penalty is the amount of time spent when there is a cache miss.

- **Block** – a fixed sized collection of data or instructions that contains the requested word that is retrieved from main memory and placed in the cache.

- **Block address** – this is the address of the block when residing in the cache.

- **Block offset** – the offset of where the word being referenced in the block.

- **Cache hit** – occurs when the referenced word from the program instruction matches with the word in the cache. The data requested is found in the cache.

- **Cache miss** – occurs when the referenced data is not found in the cache and the CPU must work its way further down the memory hierarchy to find the matching data.

- **Data Cache** – cache containing data that is frequently used and should be located close to the CPU for faster access.

- **Direct mapped** – a cache in which each block in the cache can only appear one location. This location is usually mapped as:

  location = (Block Address) MOD (# Blocks In Cache)

- **Dirty bit** – bit used to determine whether or not the block in the cache has been modified or not while in the cache. This status bit is used to reduce the frequency of writing back blocks on replacement. If the dirty bit is not set, or the block is clean, then a write back is not needed on a miss due to identical data to the cache being found at lower memory levels.

- **Fully associative** – a cache that allows the block to be placed anywhere within the cache.

- **Hit time** – the time that it takes for the hardware to transfer the data when a hit occurs on the cache.

- **Index field** – selects the set of the block in which the word being referenced points to.

- **Instruction cache** – cache containing the program instruction code that is frequently used and should be placed close to the CPU for faster access.

- **Locality** – location of the data in respect to the CPU due to how much the program is using the data. There are two types of locality we are concerned with: temporal and spatial. See *temporal locality* and *spatial locality*.

- **Memory stall cycles** – the number of CPU clock cycles in which the processor is stalled while waiting for memory access to occur.

- **Miss penalty** – the number of memory stall cycles that occurs when there is a cache miss.

- **Miss per instruction** – this is another preferred way of measuring miss rates instead of calling a miss: misses per memory reference. It is computed in the following way:

  miss per inst = (*miss rate* * *memory accesses*) / *Instruction Count*

  Miss rate is defined below, memory accesses are the number of memory accesses by the program, and the instruction count is the number of instructions executed in the program.

- **Miss rate** – this is simply the frequency of cache accesses that result in a cache miss.

- **No-write allocate** – an alternative approach to handling write misses where the block is modified in only lower-level memory and does not affect the cache.

- **Page** – fixed sized blocks in which data objects are stored in, and hopefully used in the cache.

- **Page fault** – occurs when the CPU tries to reference an item within a page that is not present in either the main memory or on disk (in virtual memory).

- **Spatial Locality** - data which is nearest the requested address or recently used data that may be used again soon, should be injected into the cache for quick access by the CPU.

- **Tag field** – bit field that is used to determine if all of the blocks are in the set.

- **Temporal Locality** - data that has been used recently will likely be used again and the least recently used data is removed while the most recently data is kept closer to the CPU.

- **Virtual memory** – hard disk space that is used as temporary memory. Due to being a hard disk access, this is the one slowest forms of computer memory.

- **Write allocate** – when a write miss occurs, a location in the cache is allocated to hold the data rather than just sending the write to a lower level in the cache-memory hierarchy for storage.

- **Write back** – the data is only written to the cache block. The block is only written back to the lower-level memory when it needs to be replaced.

- **Write buffer** – buffer to allow the processor to continue on with tasks as soon as the data is written to the cache.

- **Write stall** – when the CPU must wait for the cache write to complete during the write process.

- **Write through** – different from write back, because the data is written back to both the cache and the lower-level memory.

Measuring Cache Performance

To be able to determine the performance of the cache can really help the software engineer and compiler writers develop very good code, or maybe even the best possible code for a given problem. To measure the cache performance, we have two different measurements that can be used. The first measurement is defined as ***memory stall cycles***, which is computed as:

*memory stall cycles = number of misses * miss penalty*

This measurement is quite simple since we are counting the number of cache misses and multiplying these misses by how many clock cycles we are penalized with when a cache miss occurs. This equation can be expanded by replacing ***number of misses*** with the following:

number of misses = (*memory accesses * miss rate) / Instructions*

this will give us:

*memory stall cycles = (memory accesses * miss rate * miss penalty) / Instructions*

This last form of the original equation has more terms, but it is a bit clearer to understand than the original equation. Hennessey and Patterson [1] multiply the right side of the equation by IC, and when comparing two CPUs or two different programs for performance speedups, the IC term always cancels out and therefore is not needed.

The second approach at measuring performance of the cache is by looking at the overall execution time of the CPU. In the following equation given:

*CPU Exec Time = (CPU Clocks + Memory Stall Cycles) * Clock Cycle Time*

we are going to be making the assumptions that the amount of time needed to handle a hit on the cache is in the ***CPU Clocks*** term, and the CPU is stalled out during the event of a cache miss. Later on, we are going look at this simplified assumptions more closely.

Common Cache Questions

When introducing the cache, there are four key questions that arise and they are going to be answered in the following pages. These four common questions are:

- Where do we place a block of data in the cache?
- How do we find our block of data that was placed in the cache?
- What block should be replaced in the event of a cache miss?
- What actions occur when we write a block of data to the cache?

Where do we place a block of data in the cache?

First of all, we have three different categories in which cache are organized. These categories of cache organization are: ***direct mapped***, ***fully associative***, and ***set associative***. The direct mapped and the fully associative categorizations are described earlier in our definitions.

The ***set associative*** cache is a cache category in which the block of data to be placed in the cache is restricted to a set of places within the cache. A group of blocks in the cache is referred to as the ***set***. A block must first be mapped onto a set in the cache, then this block can then be placed anywhere within the set that it is mapped to. Computer scientist and hardware engineers often chosen the set by using ***bit selection***; and this is done by:

set = (Address of block) mod (# of sets in the cache)

If we have a set that contains more than one block, say for example, n blocks, the cache placement is then referred to as the ***n-way set associative cache***. Figure 3 shows the mapping of a *two-way set associative* cache that contains eight sets with two blocks per set and the data at block address 51 being placed in the cache. If you notice, the block can take either the first block or the second block in the set 3. We can do something like take the modulus 2 of 51 and decide which block in set 3 the block is going to actually be placed in. This happens to be similar to the direct mapped cache, except in the set associative cache, more than one block can reside in a set. The direct mapped cache, we do not have sets in the cache, and it is much simpler.

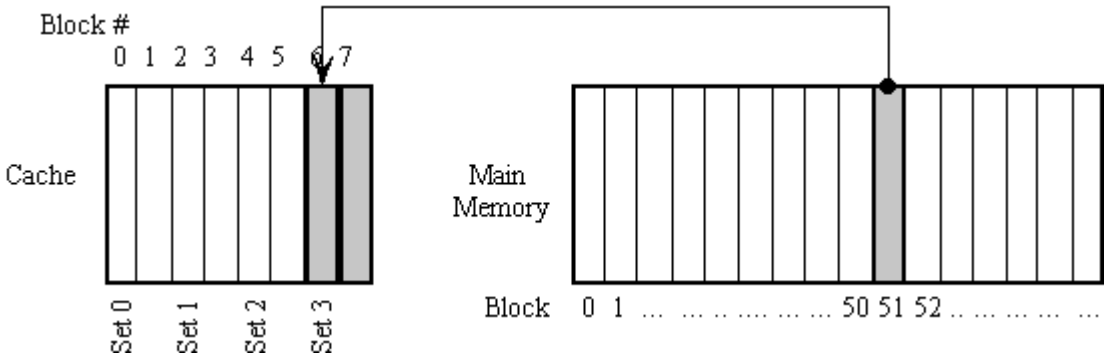


Figure 3. Example of a two-way set associative cache.

The most general categorization of a cache is the ***direct mapped cache***. In this cache, the block of data has only one place in the cache where it can be found. In a similar way like the set associative cache, we can map the location of the block by:

location = (Address of block) mod (# of blocks in the cache)

So, if we have the starting address of our block at: block address 51, and our cache contains 8 blocks, our location in the cache is going to be:

location = 51 mod 8 => 3

and our data will be stored at block 3 of the cache, as shown in Figure 4. Later on, when we need to retrieve the data associated with this memory address, we will see that we can quickly retrieve it (hopefully, something else doesn't overwrite it, such as data from address 67, which also points to 3 since 67 mod 8 => 3. But more on that later!)

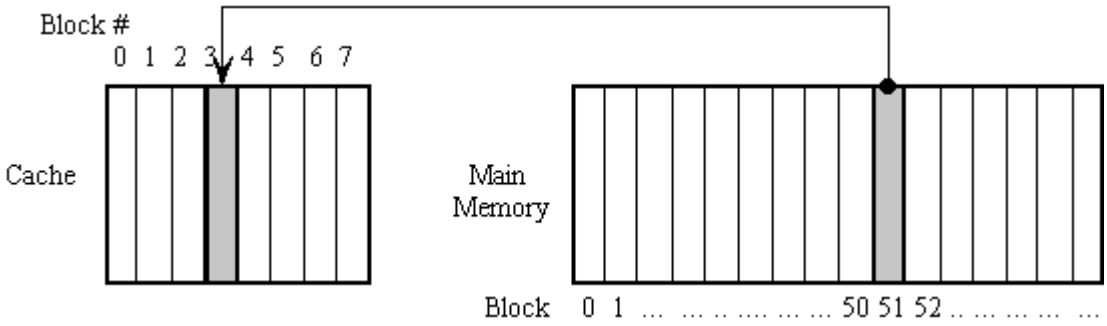


Figure 4. Example of a direct-mapped cache.

Our third category of cache organization is the ***fully associative cache***, which is a categorization of a cache in where the block of data can be placed anywhere inside the cache. The most commonly used cache categories are the set associative and the direct mapped categories since they allow for quick and easy retrieval of the data in the block when it is needed.

How do we find a block of data placed in the cache?

In order to be able to find the block of data in the cache, all blocks have an address tag that is placed on each block frame. We use this address tag to see if the block matches with the corresponding block address from the main memory. The address tags, according to Hennessey and Patterson, should be searched in some kind of parallel mechanism in order to find the block since speed is the critical factor that makes the cache so important. We also include a ***valid bit*** to test whether or not the block in the cache contains a valid address in main memory. We really do not want to have the cache work on some garbage data that does not correspond to the data in memory since this will cause uncertainties to occur in the running computer program. In Figure 5, we have a simple structure that shows us three parts of an address tag for both a set associative and direct mapped cache.

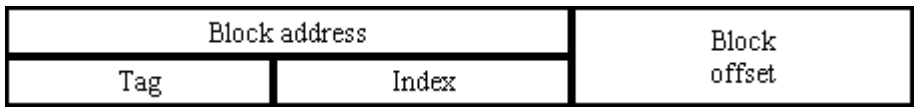


Figure 5. Address tag for set associative and direct mapped caches.

At first sight, we would think of comparing the address instead of the tag, but for two reasons, we do not need to do such a thing. First of all, the block offset should not be compared since the entire

block may or may not be present. We also could run into a problem if another address happens to have the same block offset and this could really confuse matters.

Secondly, checking the index is not very helpful at all. Since we use the index of the address tag to select the block set to be checked, we run into some redundancy when checking the address via the index field of the tag. The example given by Hennessey and Patterson [1]:

> "An address stored in set 0, for example, must have 0 in the index field or it couldn't be stored in set 0; set 1 must have an index value of 1; and so on."

With this, we can have on the computer power and speed up the hardware by reducing the width of the memory size that is used in the cache tag, and there is no need for any extra redundant hardware mechanisms where speed is so critical for making the cache something worthwhile.

What block should be replaced on a cache miss?

In the cache hardware, we have the cache controller that must select a block of data to be replaced in the event of a cache miss. We have three different approaches to handling a cache miss. These approaches are:

- Random – blocks to be replaced are randomly chosen by the cache controller .
- Least-recently used (LRU) – blocks that have been not used for a long period of time are replaced.
- First in, first out (FIFO) – the first blocks placed in the cache are the oldest.

In the random block removal approach, we are trying to uniformly spread allocation of the blocks in the cache. While this is a rather simplistic approach, the major drawback is that we could very well randomly remove a block that we may have just placed in the cache. With this happening, it could cause the number of data cache misses to increase and directly impact the performance of the program.

When we use the ***least recently used (LRU)*** approach, we begin to reduce the chance of throwing away information that we may need to access in the near future of the executing program. The idea behind the LRU approach is that if the data has been recently used in the cache, there is a chance it may be used again soon and it should be kept in the cache if this so happens that it is used again soon. The drawback with the LRU approach is that we need to use some complicated hardware approaches to implement the LRU approach.

In the final approach, the ***First in-first out (FIFO)*** approach, we do not have the complications of computing the LRU, but we can approximate it by determining the oldest block is the block that should be removed on a cache miss. Table 1 is the same as Figure 5.6 from Hennessey and Patterson [1]. It shows us a comparison between these three approaches in respect to various cache sizes.

	Associativity								
	Two-way			Four-way			Eight-way		
Size	**LRU**	**Random**	**FIFO**	**LRU**	**Random**	**FIFO**	**LRU**	**Random**	**FIFO**
16 KB	114.1	117.3	115.5	111.7	115.1	113.3	109.0	111.8	110.4
64 KB	103.4	104.3	103.9	102.4	102.3	103.1	99.7	100.5	100.3
256 KB	92.2	92.1	92.5	92.1	92.1	92.5	92.1	92.1	92.5

Table 1. Data Cache misses per 1000 instructions from Hennessey and Patterson.

When we take a look at random block removal approach, we see that the highest miss rates occur when the cache is smaller. When we increase our associativity, it begins to lower. The reason behind this is because we are taking out data blocks that have been recently used and are needed again, hence there is a good deal of cache thrashing occuring with the random block removal approach. The random block removal approach miss rate begins to lower as the associativity and the size of the cache increases. This is because there is less data block thrashing that is taking place on the cache.

Observing the LRU and FIFO approaches, it seems that the LRU approach has a better miss rate than the FIFO approach by anywheres between 0.33% and 1.41%. The LRU approach removes the least recently used block, where as the FIFO removes the oldest block. The LRU even with its complexity in hardware, is making sure that the block that hasn't been accessed for some time is going to be replaced under the presumption that it is not going to be accessed again soon. In the FIFO approach, the first block (and oldest) may still be being accessed on occasion by the program, hence when this old block is removed, and all of the sudden it is needed again, it has to be reloaded into the cache because of the cache miss. This explains why its miss rate is just a little higher than the LRU approach. Notice that the n-way associativity is not a factor for either of these approaches, but the size of the cache is still a factor in which the larger the cache, the miss rate is lower.

What goes on when we write a block to the cache?

When writing a data block to the cache we have to choose between doing a ***write-through*** or a ***write-back***. These two options determine how information is going to be written. In the *write-through*, we are going to write the information back to both the cache block and to the block in lower main memory (this could also be another level in the cache, such as the L2 cache if we are working on the L1 right now). In the *write-back* approach, the data is only written back to the cache and it is only written back to main memory when the block is replaced in the cache by one of the block removal approaches just described in the last section.

Part III. Evaluating Cache Performance

To evaluate the cache performance, we need to look at two different approaches at evaluating the cache performance. The first approach is to analyze the average memory access time and processor performance. The second approach for evaluating the cache performance is to analyze the performance with cache miss penalty and out-of-order execution processors. The first approach is an easy one to look at, where as the second approach can be a bit more tricky. According to Hennessey and Patterson [1], it is a matter of question whether or not the average memory access time due to a cache miss has any correlation to processor performance.

Let us assume that we have an Intel Pentium 4 that has a miss penalty of 25 clock cycles per miss, has a typical CPI of 1.75, an average miss rate of 3%, the running program has an average of ½ memory references per instruction (this is every other instruction having a memory reference, this is quite typical in many cases). Knowing information like this can provide us with a way to answer the following questions: what is the performance of the CPU when the cache is included?, and what is the cache performance when misses per instruction and miss rate are both included?

With the first question, we would compute:

$$CPU_{with\ cache} = 1.75 + (30/1000)\ x\ 25 = 2.50\ cpi$$

As far as with the second question, we would compute:

$Cache_{with\ miss\ rate} = 1.75 + (0.5 \times 0.03 \times 25) = 2.13\ cpi$

If we were to disregard the cache memory hierarchy, we can compute the amount of speed up using the cache as followed:

$Speedup_{with\ cache} = 1.75 + 25 \times 0.5 = 14.25\ cpi.$

This shows us how important it is to have a cache included as an important part of the computer hardware. As a side note, I have gone into the BIOS on my 2.53ghz machine, disable the L2 cache, and Windows XP Professional went from booting in 25 seconds (with L2 cache enabled) to an excess of 10 minutes (with L2 cache disabled)!

Part IV. Improving Cache Performance

There are three areas at improving cache performance. These areas are: reducing cache miss penalties, reducing the miss rate, and the reduction of hit time on the cache.

Reducing Cache Miss

There are four approaches to reducing cache miss penalties in order to increase the performance of the cache. These approaches are: the use of multi-level caches, critical word first and early restart, write buffer merging, and use of victim caches.

Using Multilevel Caches

The computer architect has two questions that they must answer: should cache be faster in order to keep up with the CPU speed?, or should the cache size increase to overcome the gap between the CPU and the main memory? The computer architect needs to do both and the architect does so by adding multiple levels to the cache. In this tutorial, we are going to be working with the L-1 cache on the CPU, and the L-2 cache is assumed to be in between the CPU and main memory. The L-1 cache is the smallest and the fastest of any of the levels of cache, so it is able to keep up with the CPU speed. The L-2 cache can be significantly larger than the L-1 (in most machines, the L-1 is generally about 16-64KB and the L-2 ranges from 128-512KB as of this writing), and a bit slower than the L-1. However, the L-2 is able to keep a fairly good sized chunk of data close to the CPU in case the CPU is going to need it within a short time-frame. This helps reduce the miss penalty between the L-1 and L-2, and also reduces the miss rate between the L-2 and main memory (hopefully the machine doesn't have to go far down the cache hierarchy to find the block).

With a cache system that has two or more levels of cache, we have both the local and global miss rates. In the equation below, the local miss rates are: $Miss\ rate_{L1}$ and $Miss\ rate_{L2}$ for their respective cache level. However, the global miss rates are: $Miss\ rate_{L1}$ for the L-1 cache, and $Miss\ rate_{L2} \times Miss\ rate_{L1}$ for the L-2 cache.

$$avg.\ memory\ access\ time = Hit_{L1} + Miss\ rate_{L1}\ (Hit_{L2} + Miss\ rate_{L2} \times Miss\ Penalty_{L2})$$

This equation is used when we want to find what the average memory access time is when using a multi-level cache with the L-1 and L-2 cache. It is helpful when we are trying to improve the

performance of the cache. However, the next equation is a much more helpful equation since it deals with how many clock cycles the computer stalls when there is a cache miss on an instructional basis.

$$\textit{avg. memory stalls per instruction} = \textit{Miss per inst}_{L1} \times \textit{Hit time}_{L2} + \textit{Miss per inst}_{L2} \times \textit{Miss penalty}_{L2}$$

In the following example, we can compare these two equations and see which of these two equations can help us get a good picture of the cache performance. If we have 1,000 memory references with the miss rates of 30 misses on the L-1 cache, 15 misses on the L-2 cache, there is a penalty of 50 clock cycles for a miss on the L-2, and we have a hit time on the L-1 cache of 1 clock cycle and 8 clock cycles on the L-2, what are the various miss times that we could be facing?

avg. memory access time = 1cc + 0.03 (8cc + 0.50 x 50cc) = 2 clock cycles
(0.50 = 15/30)

avg. memory stalls per instruction = (0.03 x 8) + (0.15 x 50) =7.74 clock cycles

Bear in mind that neither of these two equations are necessarily better than the other. They both can tell us how well the multi-level cache is performing. When we think about it, on a miss, we could be looking at up to 9.74 clock cycles going to waste when there is a miss on the cache.

Critical Word First and Early Restart

The Critical Word First and Early Restart approach is based on the CPU needing only one word from the block at a time. The CPU requests the missed word from memory and works with the missed word while the remaining words of the block are loaded into the cache during the instruction execution, and this is called Critical Word First. If we decide to load all of the words of the block in a normal manner and when all of the blocks are loaded into the cache, this is Early Restart. The cache is sent to the CPU and the CPU continues with code execution. Critical Word First and Early Restart is best for architecture design in which there is a large cache. The only drawback with this approach is the spatial locality between the cache and main memory.

Write Buffer Merging

The key to the Write Buffer Merging approach is to increase the efficiency of the write buffers. Since write-through caches depend on write buffers, all of the data stores must be passed down to the next lower level in the memory hierarchy. Figure 6 illustrates write buffer merging.

Write Address	V		V		V		V	
1000	1	Mem[1000]	0		0		0	
1004	1	Mem[1004]	0		0		0	
1008	1	Mem[1008]	0		0		0	
1012	1	Mem[1012]	0		0		0	

Write Address	V		V		V		V	
1000	1	Mem[1000]	1	Mem[1004]	1	Mem[1008]	1	Mem[1012]
	0		0		0		0	
	0		0		0		0	
	0		0		0		0	

Figure 6. Using Write Buffer with merging is on the top and Write Buffer without merging is on the bottom.

The buffer has four 32-bit word entries. The address for each entry is on the left and the valid bit denoted by V indicated whether or not the next 4 bytes are occupied in this entry.

Victim Caches

The last approach for reducing the cache miss penalty is to utilize victim caches. We use the victim cache to reduce the miss penalty by holding onto recently discarded data in case we need to request it again. Since this discarded data has already been fetched by the CPU, there is little cost in regards to re-using the data again. In figure 7, we have a fully associated cache and a refill path, which makes up the basics for a victim cache.

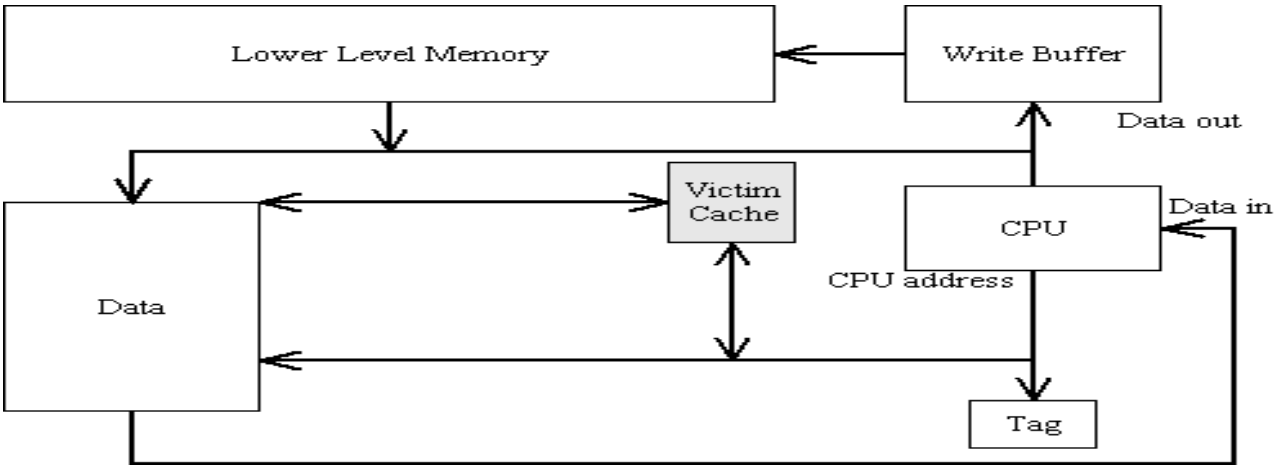


Figure 7. Victim Cache

We check the contents of the victim data (data that was removed from the cache) in the victim cache against the blocks that are going to be possibly fetched. If there is a match between the blocks that may be fetched and the victim data, then the blocks in the victim cache and the regular cache are swapped. The more effective victim caches have between one and five entries. The AMD Athlons use an eight entry victim cache and this can in theory remove 1/8 of the misses on the cache.

Miss Rate Reduction

Perhaps the most common way to improve on the cache is by reducing the amount of misses on the cache. This is miss rate reduction. I will be discussing four approaches for reducing the cache miss rate. Types of cache misses are categorized by what is known as the "three C's." These categories are: *compulsory, capacity,* and *conflict*.

Larger Block Sizes

One of the easiest ways to reduce cache miss rate is by increasing the size of the blocks. By utilizing a larger size, the amount of compulsory cache misses is lower. The primary reason for the lower miss rate is due to larger blocks taking advantage of spatial locality. The only drawback to a larger block size is that there may be an increase in conflict misses and/or capacity misses if the cache is too small. Care must also be taken to ensure there is no dramatic increase in average memory access time over the miss rate reduction. Table 2 shows SPEC2000 traces from Hennessey and Patterson. One

can take note that for caches that are less than 64K in size, 32-64 byte blocks tend to have the better miss rates, however, when the cache exceeds 64K in size, the larger blocks are much better.

	Cache Size			
Block Size (bytes)	**4K**	**16K**	**64K**	**256K**
16	8.57%	3.94%	2.04%	1.09%
32	7.24%	2.87%	1.35%	0.70%
64	7.00%	2.64%	1.06%	0.51%
128	7.78%	2.77%	1.02%	0.49%
256	9.51%	3.29%	1.15%	0.49%

Table 2. Miss rates compared to Block/Cache sizes from Hennessey & Patterson.

Larger Caches

One would quickly conjure that the better way to reduce miss rates along with reducing capacity misses at the same time, is to increase the overall size of the cache. For the most part, this is true, however, larger caches introduce the drawback of longer hit times on the cache. Because of the longer hit times, a large cache on-chip would not be quite advantageous. Larger caches are better for off-chip caches such as the L-2 cache, and this is why when we look at CPUs, we have smaller 16-32KB cache on chip and anywhere between 256-512KB off chip.

Compiler Optimization Techniques

All of the techniques for improving the cache performance discussed up to this point all have one thing in common. The common characteristic between all of these techniques is that the software developer really does not have much control over cache performance. However, the software developer can improve the cache performance through optimization techniques that can help data be loaded into the cache in a more optimal manner. A common optimization techniques are *loop interchanging*, .

Loop Interchanging

In just about any program that is written, at some point there are going to be nested loops. Unknown to the amateur programmer, the order in which loops are nested can have an incredible impact on the code performance in the cache. In Appendix A, there is the program *loop_interchange.cpp* which has a #define A_SIZE of 1000 (this can be changed at will), and the size of the two dimensional array that is being worked on is A_SIZE by A_SIZE * 1.5 (in the default case, 1000 x 1500). The general pseudocode for loop interchange is the following:

```
For I = 1 to 1000 do
        For J = 1 to 1500 do
                A(I,J) = 2 * A(I,J)
        End do
End Do
```

The problem with this pseudocode is that the code will run through the array in strides of 1000 machine words. This will cause a tremendous amount of cache thrashing due to having to reload new lines of data at each iteration. In the next pseudocode listing, the I and J loops are simply interchanged to prevent the code from skipping through memory in 1000 machine word chunks, thus reducing unnecessary cache loads.

```
For J = 1 to 1500 do
        For I = 1 to 1000 do
                A(I,J) = 2 * A(I,J)
        End do
End Do
```

In Part V, we will look over the test benches and analyze the results of this technique.

Matrix Multiplication with Cache Blocking

In many applications such as 3D graphics rendering, economic forecasting with linear systems, and many more, matrix multiplication is a very important aspect in these areas. The general matrix multiplication algorithm that is taught in linear algebra courses is good for most people to get the general idea of how matrices are multiplied. However, when it comes to high performance computing, the general matrix multiplication algorithm does not always work well with the cache. As long as the matrices can fit in the cache, this is fine. When we are dealing with matrices that are so large that fitting them into the cache is impossible, we need to resort to a matrix multiplication algorithm with cache blocking. In this algorithm, we break the matrices down into smaller sub-matrix blocks that can be handled by the cache. In Part V, we will look over test benches and analyze the effects of cache blocking matrix multiplication on the cache. The code is available in Appendix A.

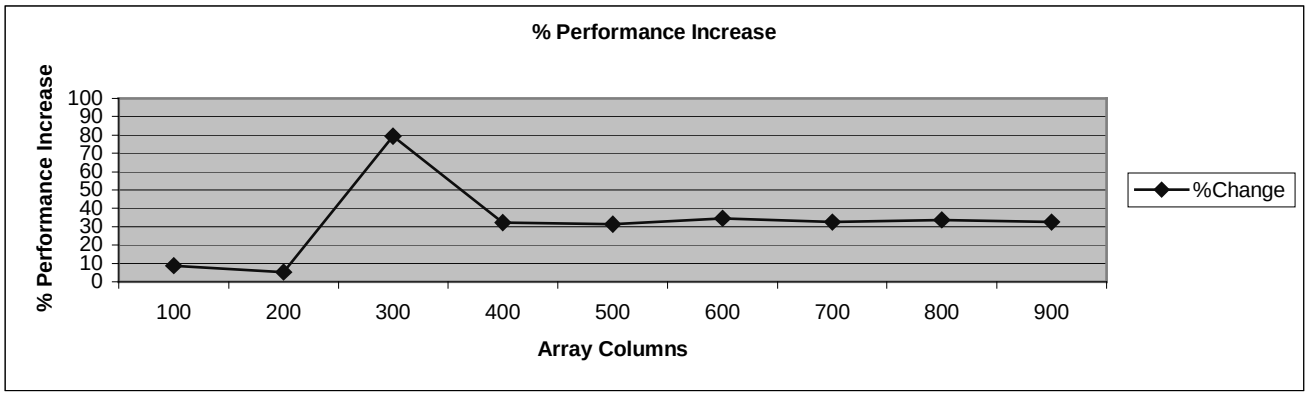
Loop Unrolling to Reduce Cache Data Reloads

In a similar context of using loop interchange to reduce the amount of data reloads, we can apply the same with loop unrolling. In this technique, we simply unroll loops to fit blocks of data into the cache per iteration of the loop to allow for cache misses to be reduced. However, it must be exercised with caution that unrolling a loop too much can have a negative impact on the cache performance. As with all of these techniques, testing and fine-tuning can help the cache performance be optimal under the given conditions.

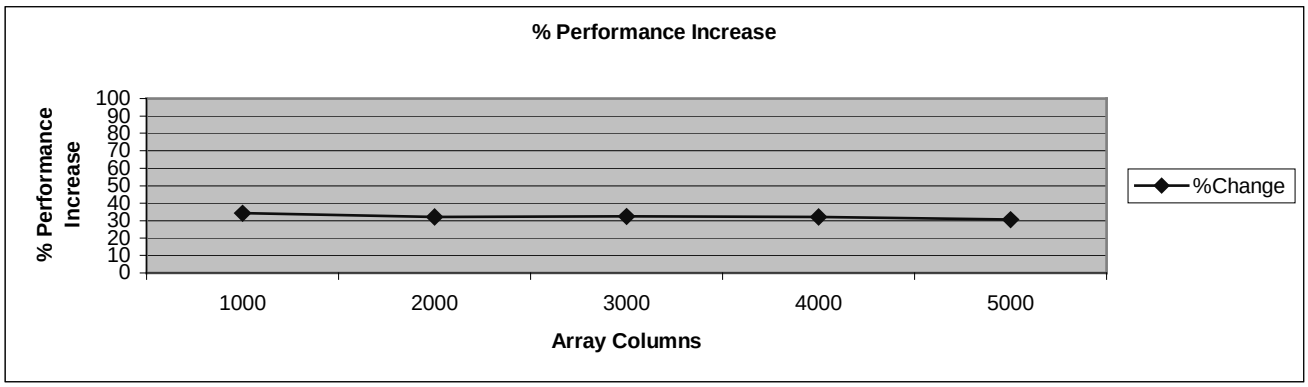
Part V. Cache Test Benches and the Results

Loop Interchanging Results

In the loop interchanging technique of the compiler code optimization, it was found that for very small arrays such as 100x150 and 200x300 (5 to 10% increase range), the performance increase was not substantial. However, it was found that the performance increase exceeded 32% for arrays that were 400x600, 500x750, 600x900, 700x1050, 800x1200, 900x1350, and 1000x1500. The peak performance increase was the 300x450 array with 79% performance increase on a Pentium 4 with 512KB L-2 cache.



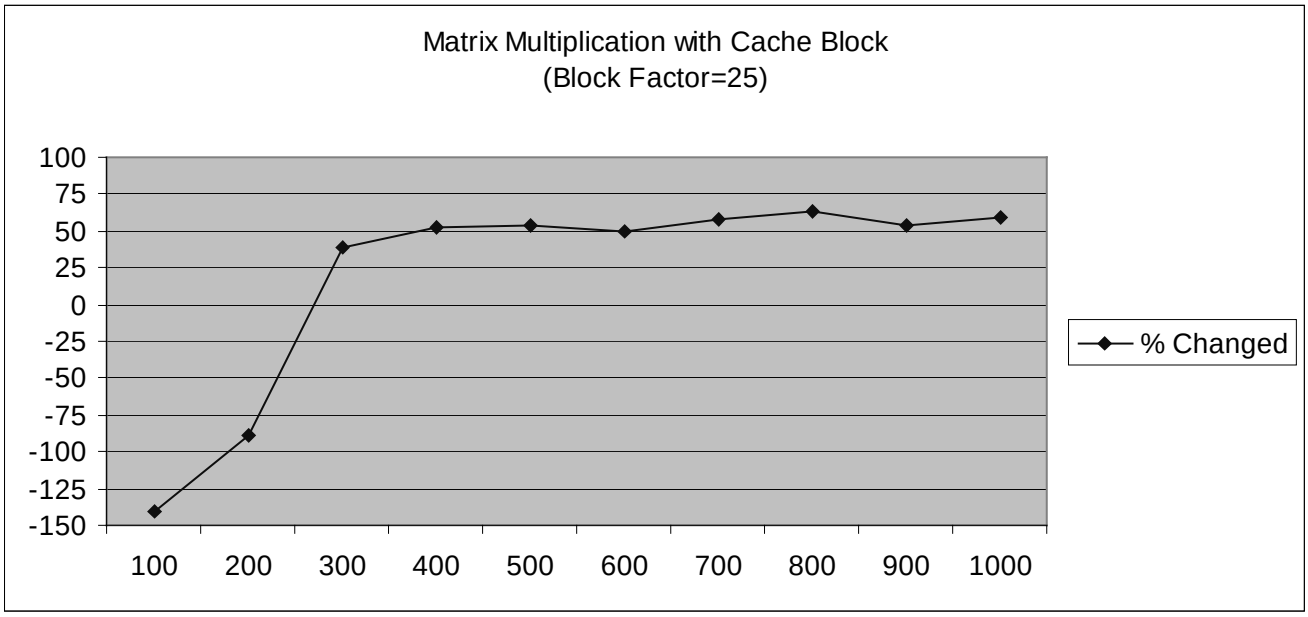
In a second test of the loop interchange, it was tested in increments of 1000's instead of 100's like in the previous test. The results stayed within the 30-34% range with a slight decrease in performance increase, which was probably due to beginning use of virtual memory.



Conclusion is that the loop interchanging technique for compiler optimization is indeed a good way of making the cache performance increase worthwhile. The technique is an easy technique to apply, whereas the next area that is going to be analyzed is not as easy and straight forward as the loop interchange approach.

Matrix Multiplication with Cache Blocking Results

In the matrix multiplication with cache blocking, tests were done with the blocking factor set at 25 since this would yield three (3) matrices using 2,500 bytes each for a total of 7.3KB of the L-1 cache being used. Twenty-five (25) also is the largest number that divides cleanly with multiples of 100's (i.e. if 30 were used, it still fits in the L-1 cache fine, but does not really evenly divide for the most part). The wcpuid utility, which can be found on the web, states that the test machine has 12KB instruction and 8KB data on the L-1 cache. The algorithm does not really start to become beneficial until matrices of 300x300 are multiplied. This would be due to the blocking algorithm taking more time working with setting up blocks than the traditional matrix multiplication algorithm. The traditional matrix multiplication algorithm begins to strain the cache (especially the L-2) at matrices of 300x300 and larger, whereas the cache blocking algorithm always keeps within the constraints of the L-1.



It can be concluded that for smaller instances of matrix multiplication, the original algorithm is still good because it can fit in the cache, but with the modern cache capacities, there is not much in terms of headroom for larger matrices. This is where cache blocking truly becomes beneficial. Though out of the scope of this tutorial, block cache on distributed systems can further increase the performance of matrix multiplication with the cache blocking algorithm when done in parallel.

Part VI. Conclusions

As mentioned earlier, the key to having a cache hierarchy is to allow for programs to execute commonly used code at a much faster rate. The test computer did have the L-2 cache disabled and it took about 5 minutes to boot Windows XP Pro™ with the cache disabled. The loop interchange program was also executed with test sizes between 100 and 1000 with varied results. One instance had an 80% speed increase and two instances had more than –8% increase (in which the original nested loop fared better). However, it must be kept in mind that clock cycles were counted and there are substantial differences. For loop interchanging using 100x150 array, the cached run execute in 41,900 clock cycles and the non-cached execution took 606,080 clock cycles. Table 3 has a nice compare and contrast between these two test runs.

	Cache Disabled			Cache Enabled				Speedup	
N	Before	After	% Change	Before	After	% Change		Before	After
100	557516	606080	-8.71	45867.2	41913.6	8.69		12.2	14.5
200	2171892	1912504	11.94	182463.2	172664.8	5.36		11.9	11.1
300	4880588	4503532	7.73	1885248.8	390027.2	79.31		2.6	11.5
400	8683228	8318428	4.20	6688012.8	4653609.6	32.35		1.3	1.8
500	13585040	12415816	8.61	10269674.4	7028090.4	31.39		1.3	1.8
600	24171060	18186212	24.76	15168527.2	9874004	34.58		1.6	1.8
700	26772404	24083244	10.04	20288327.2	13497925.6	32.58		1.3	1.8
800	37041392	40345080	-8.92	26761744.8	17738438.4	33.67		1.4	2.3
900	151023056	41617636	72.44	33893680	22816770.4	32.68		4.5	1.8
1000	57340480	50272272	12.33	41842148.8	27533932	34.10		1.4	1.8
							Avg	3.9	5.0

Table 3. Cache vs no cache compare and contrast.

As we can see, in most of the cases in the *after* column, there are increases between 180 to 1450% when the cache is used. It is without a doubt that the cache is an important part of being able to compute large calculations and manipulate large amounts of data within a reasonable amount of time. This is why it is important for software developers to know about the architecture that they are working with. It helps computer scientist fully utilize the machine. As a close, I hope this tutorial is able to give aspiring computer scientists a good foothold when it comes to writing excellent code that can use a processor to its maximum.

Part VII. Appendix A: Program Listings

```
/*
   loop_interchange.cpp
   Paul Conrad
   July 2004
*/

#include <iostream>
#include <math.h>
#include <fstream>

using namespace std;

#define B_SIZE (int)(1.5*A_SIZE)

int array[B_SIZE][A_SIZE];

/* Utilize the rdtsc instruction from Pentium class processors
   Input:  none
   Output: Clock cycles elapsed since computer started
*/
__int64 readTSC() {
      __asm { rdtsc };
}

/* loop_interchange entry point
   Input:  none
   Output: 1 - no real reason.
*/
int main() {

      // Clock cycles before and after code executed
      __int64 before_start, before_cc;
      __int64 after_start, after_cc;

      // Fill in two dimensional array with values
      for(int j=0;j<B_SIZE;j++) {
            for(int k=0;k<A_SIZE;k++) array[j][k]=j+k+1;
      }

      // Time code with first nested loops
      before_start=readTSC();
      for(int j=0;j<B_SIZE;j++) {
            for(int k=0;k<A_SIZE;k++) array[j][k]=2*array[j][k];
      }
      before_cc=readTSC()-before_start;      // Total Clock Cycles executed

      // Refill two dimensional array with values since in last code block
```

```
	// data was destroyed.
	for(int j=0;j<A_SIZE;j++) {
		for(int k=0;k<B_SIZE;k++) array[j][k]=j+k+1;
	}
	after_start=readTSC();

	// Time code with interchanged loops
	for(int j=0;j<A_SIZE;j++) {
		for(int k=0;k<B_SIZE;k++) array[j][k]=2*array[j][k];
	}
	after_cc=readTSC()-after_start;	// Total Clock Cycles executed

	// Compute the speedup (if any)
	float speedup=100*((float)(before_cc-after_cc)/(float)before_cc);

	// Display results to terminal screen
	cout<<"'Before' Loop Interchange:  "<<before_cc<<" clock cycles."<<endl;
	cout<<"'After' Loop Interchange:  "<<after_cc<<" clock cycles."<<endl;
	cout<<endl;
	cout<<"Speedup:  "<<speedup<<"%"<<endl;

	// Also put results in a comma separated values file to allow
	// data manipulation from within a spreadsheet.
	fstream output_csv("loop_interchange_log.csv",ios::out|ios::app);
	output_csv<<A_SIZE<<","<<before_cc<<","<<after_cc<<","<<speedup<<endl;

	// Bail out
	return 1;
}
```

Note: use /DA_SIZE=*num* where *num* is the number of rows in the array. The number of columns will be 1.5**num*.

```
/*
   matrices.cpp
   Paul Conrad
   July 2004
*/

#include <iostream>
#include <math.h>
#include <fstream>
using namespace std;

int a[A_SIZE][A_SIZE], b[A_SIZE][A_SIZE], c[A_SIZE][A_SIZE], corr[A_SIZE][A_SIZE];

/* Find the minimum of two integers
   Input:  integers a, b
   Output: the smaller integer of a or b
*/
int MIN(int a, int b) {
	return a <= b ? a : b;
}

/* Utilize the rdtsc instruction from Pentium class processors
   Input:  none
   Output: Clock cycles elapsed since computer started
*/
__int64 readTSC() {
	__asm { rdtsc };
}

/* Clear the resultant matrix
   Input:  none
   Output: none
*/
void clearMatrix() {
	for(int u=0;u<A_SIZE;u++) {
		for(int y=0;y<A_SIZE;y++) c[u][y]=0;
	}
}

int main() {
	__int64 before_start, before_cc;
	__int64 after_start, after_cc;
	int iterate;

	// Initialize matrices a and b
	for(int j=0;j<A_SIZE;j++) {
		for(int k=0;k<A_SIZE;k++) {
			a[j][k]=j+k+1;
			b[j][k]=j+k-1;
		}
```

```
	}

	// Make sure resultant matrix is cleared
	clearMatrix();

	// Matrix multiplication without blocking
	before_start=readTSC();
		for(int j=0;j<A_SIZE;j++) {
			 for(int k=0;k<A_SIZE;k++) {
				for(int l=0;l<A_SIZE;l++) c[j][k]+=a[j][l]*b[l][k];
			}
		}
	before_cc=(readTSC()-before_start);	// Total Clock Cycles

	// Make sure resultant matrix is cleared for next test
	clearMatrix();

	// Matrix multiplication via cache blocking
	after_start=readTSC();
	int ii, kk;

	for (ii=0; ii<A_SIZE; ii+=25) {
		for (kk=0; kk<A_SIZE; kk+=25) {
			for (int j=0; j<A_SIZE; j++) {
				 for (int i=ii; i<MIN(ii+24,A_SIZE); i++) {
					for (int k=kk; k<MIN(kk+24,A_SIZE); k++) {
						c[i][j] += a[i][k]*b[k][j];
					}
				}
			}
		}
	}
	after_cc=(readTSC()-after_start);	// Total Clock Cycles

	// Compute the speedup (if any)
	float speedup=100*((float)(before_cc-after_cc)/(float)before_cc);

	// Output to terminal
	cout<<"Regular Matrix Mult:  "<<before_cc<<" clock cycles."<<endl;
	cout<<"'Blocking' Method (block factor=25:  "<<after_cc<<" clock cycles (";
	cout<<"Speedup:  "<<speedup<<"%"<<endl;

	// Output to comma separated value log file for data
	// manipulation within a spreadsheet
	fstream output_csv("matrices_log.csv",ios::out|ios::app);

	output_csv<<A_SIZE<<","<<before_cc<<","<<after_cc<<","<<speedup<<endl;

	return 1;
}
```